Applied and
Computational
Mathematics
Division

NISTIR 5569

Computing and Applied Mathematics Laboratory

The MasPar MP–1 as a Computer Arithmetic Laboratory

M. A. Anuta, D. W. Lozier
and P. R. Turner

January 1995

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

NIST

NISTIR 5569

The MasPar MP-1 as a Computer Arithmetic Laboratory

M. A. Anuta
D. W. Lozier
P. R. Turner

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Applied and Computational Mathematics
Division
Computing and Applied Mathematics
Laboratory
Gaithersburg, MD 20899

January 1995

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

The MasPar MP-1 as a Computer Arithmetic Laboratory

Michael A Anuta[1], Daniel W Lozier[2] and Peter R Turner[3]

Abstract

This paper describes the use of a massively parallel SIMD computer architecture for the simulation of various forms of computer arithmetic. The particular system used is a DEC/MasPar MP-1 *with* 4096 *processors in a square array. This architecture has many advantages for such simulations due largely to the simplicity of the individual processors. Arithmetic operations can be spread across the processor array to simulate a hardware chip. Alternatively they may be performed on individual processors to allow simulation of a massively parallel implementation of the arithmetic. Compromises between these extremes permit speed-area trade-offs to be examined. The paper includes a description of the architecture and its features. It then summarizes some of the arithmetic systems which have been, or are to be, implemented. The implementation of the level-index and symmetric level-index, LI and SLI, systems is described in some detail. An extensive bibliography is included.*

I. Introduction

This paper describes and discusses the use of a massively parallel SIMD computer system as a computer arithmetic laboratory. Specifically the DEC/MasPar MP-1 system with 4096 processors is used for software implementation of various types of computer arithmetic for integer, fixed-point, real and complex arithmetic. The systems implemented (or, in some cases, to be implemented) include both conventional and novel representation and arithmetic systems. Some of these provide general computational frameworks (such as binary integer and floating-point). Others have been developed primarily as special arithmetic systems (such as the RNS) or are still in experimental design stages (such as logarithmic, level-index and symmetric level-index arithmetic).

The first part of the paper contains a brief introduction to the MasPar architecture and why it is appropriate for this task. Section 3 reviews some of the number representations and their corresponding arithmetic data types which have been (or, in some cases, are being) created in this laboratory. In Section 4, we concentrate on one particular case. The implementation of the symmetric level-index, SLI, arithmetic serves as a particularly illustrative example of the general laboratory project because it uses some of the other arithmetic systems (such as fixed point fractional arithmetic of various wordlengths) for its internal

---

[1] Cray Research Inc, 4041 Powder Mill Road, Suite 600, Calverton, MD 20705

[2] Applied & Comp Math, National Institute of Standards & Technology, Gaithersburg, MD 20899

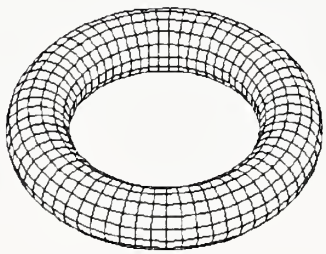
[3] Mathematics Department, United States Naval Academy, Annapolis, MD 21402

processing. This section also contains details of a modified algorithm for SLI arithmetic which is better suited to a massively parallel implementation - and to an eventual VLSI hardware implementation of SLI arithmetic. A substantial bibliography is included.

II. The MasPar MP-1 System

The MasPar system used is a SIMD array of 4096 processors configured as a square 64×64 array with toroidal wraparound in both directions. The individual processors are just 4-bit processors so that all arithmetic is implemented in microcode. Like any SIMD architecture, at any instant all processors are either performing the same instruction or are *inactive*. Clearly, for example, adding two 64×64 matrices is a particularly simple instruction for this machine. Matrix multiplication is less straightforward but is still well-suited to the array. Its speed advantage relative to conventional architectures comes from the massive parallelism overcoming the slower individual operations.

FIGURE 1
Conceptual diagram of SIMD array with toroidal wraparound



The principal advantages of using such a SIMD array for the implementation of a computer arithmetic laboratory arise out of its flexibility.

The 64×64 array of four-bit processors can be used to simulate hardware implementations of the various arithmetic schemes and to make alterations easily in the algorithms being used. Alternatively arithmetic can be implemented using serial algorithms so that the main computation is spread across the processors. This allows computation to take advantage of the parallelism to reduce the time-penalty inherent in such a software system.

By implementing the standard floating-point and integer arithmetic in a similar manner, it should be possible to create a "level playing field" for comparing the performance of different arithmetic systems on particular problems. In particular, timing comparisons can be made fairly easily since even the built-in arithmetic is "nibble-by-nibble". Since a 4-bit nibble corresponds to a hexadecimal digit, using radix 16 to implement the internal arithmetic of any system is natural. This also has the benefit that even multiplications can then be accommodated by using the built-in 8-bit integer format as a basic building block.

The parallel array will allow realistic experimental computation without the enormous time-penalties which would be suffered on conventional serial machines - or even on pipelined vector processors.

By making a compromise between the "spread-the-arithmetic-across-the-array" paradigm and the "serial-algorithm-executed-in parallel" alternative, speed-area trade-off simulations can be run. The relative performances can be expected to be reasonably indicative and so to alleviate the need for building experimental hardware.

Many of these points will become clearer with reference to particular implementations later.

The MP-1 programming languages are MPL and MPFortran. MPL is an extended version of ANSI C allowing for *plural* variables which are variables for which there is an instance on each processor - or, more precisely in each processor's individual memory. Communication between the various processors and their memories is achieved either through the *Xnet* (which is designed for nearest-neighbor communication in each of the North, South, East and West directions) or the *router* which handles more distant communications.

MPFortran is a version of high-performance Fortran which again includes the appropriate array constructs and communication instructions. The two languages have been designed for the easy inclusion of subroutines written in one language within programs in the other. There is also a very powerful debugging and program-development environment which includes a profiler so that bottlenecks are easily identified. An additional advantage is that these languages will lead fairly naturally to more portable code in F90 or C++.

A later phase of the development of this computer arithmetic laboratory will be the simulation of various arithmetic hardware components. Then a prospective chip design could be mapped onto the array and tested.

III. Review of proposed computer arithmetic systems

Integer and floating-point arithmetic already exist in both MPL and MPF in all the standard wordlengths, and some others such as a built-in *long long* which is a 64-bit integer in MPL. Complex floating-point arithmetic is also available in MPF. This section contains a brief summary of some of the other formats which are (or will be) available in the computer arithmetic laboratory. The list is merely illustrative and is not intended to be complete.

III.1 Integer and fixed-point arithmetics

III.1.1 Binary integer arithmetic

Standard (two's complement) binary integer arithmetic already exists on the MP-1 and so need not be implemented specially for this laboratory. Indeed the standard and, especially, the shorter wordlength integer forms will be used as a basis for many of the other implementations. At a later stage of the development many of the hardware components of binary integer processors will be simulated to assist with the design of hardware algorithms. Details of these algorithms are readily available in standard texts such as [1]-[7] Online algorithms, signed digit and redundant arithmetic ([8]-[13], for example) are often used for the internal computation. These would also be implemented during this later stage.

III.1.2 Fixed-point fraction arithmetic

One of the arithmetic forms which is often missing from the usual computational data types is fixed-point fraction arithmetic. Systems such as the lexicographic continued fractions of Kornerup and Matula [49]-[53] provide a general rational arithmetic. Otherwise, typically, binary fixed-point fractions are implemented as scaled versions of integers.

The fraction arithmetic implemented within this computer arithmetic laboratory allows direct computation with fixed-point fractions of varying wordlengths. Specifically, the wordlength is measured in "nibbles" (or hexadecimal digits). One nibble is reserved for sign and other information - such as a record of overflows for addition or the use of a reciprocation bit in division.

Fraction arithmetic is often required not only for itself but also for the internal computation of other arithmetic representations such as the level-index scheme which is discussed in greater detail in the next section. Some of the details of the implementation of fraction arithmetic are also presented there.

The use of the "nibble-base" means that multiplication of digits can be easily performed in an 8-bit integer format. Division is readily implemented using a radix-16 nonrestoring algorithm.

The basic fraction arithmetic is also to be extended for various library functions including some special function definitions which are needed for efficient algorithms for LI, SLI or logarithmic arithmetic. These arithmetic algorithms also require the use of fixed-point number representations which have both an integer and a fraction part. These representations are accommodated by allowing "fractions" with $n.m$ hexadecimal digits meaning n integer digits and m fraction nibbles.

III.1.3 RNS arithmetic

Residue Number Systems (RNS) arithmetic has been extensively researched for well over twenty years and there is a very considerable literature on the representation, arithmetic algorithms and applications of such systems. A sample of these are listed in the Residue Number Systems section of the Bibliography, [14] - [28]

The principle of RNS arithmetic is that an integer within the representable range is represented by its residues modulo a set of basis primes. (Strictly, not all the basis elements must be prime but for most practical purposes this is needed.) Thus an integer N is represented in the RNS system using base moduli $p_1, p_2, \ldots, p_L$ by the vector $(a_1, a_2, \ldots, a_L)$ where

$$a_i \equiv N \bmod p_i \qquad (i = 1, 2, \ldots, L)$$

Addition and subtraction of integers represented in this way can be performed by adding (or subtracting) the respective residues - and this may be done entirely in parallel since there is no carry from one modulus to another. The same is true for multiplication provided that the product does not overflow the *dynamic range*

$$M = \left(\prod_{i=1}^{L} p_i \right) - 1$$

(For many practical applications of RNS arithmetic, a symmetric range equivalent to $[-M/2, M/2]$ would be used.)

The implementation of RNS arithmetic on the MP-1 would use one processor per modulus. Usually, the dimension L of the RNS-basis is much smaller than the 4096 processors available and so it becomes feasible to implement a high degree of SIMD parallelism at the conventional level. For example even with a RNS-basis of 64 moduli, the MP-1 can simulate a SIMD vector processor with 64 processors each operating on this extended data type.

The implementation covers the common RNS integer arithmetic formats - both the nonnegative and symmetric forms. Conversion of either of these to binary integer forms can be achieved using the Chinese Remainder Theorem, CRT. The processor array can be used to implement the long accumulator which is needed for this conversion with a large dynamic range.

Other features which are included are base extension using a mixed radix conversion and the quadratic extensions of RNS integer arithmetic to admit complex integer arithmetic. Both the "real and imaginary part" form of the QRNS and the logarithm-based GEQRNS (Galois-enhanced quadratic residue number system) are implemented. (See [23] for example.)

Various RNS division algorithms have been (or will be) included for comparison purposes. These include the newer algorithms of [20] and [27]. One of the first applications of this arithmetic will be to the solution of linear systems and, in particular, the adaptive beamforming problem.

III.2 Real number representations and arithmetic

III.2.1 Floating-point systems

The standard IEEE floating-point data types are provided in MPL and MPF. The laboratory will include software implementations of these with variations to allow for different wordlengths and different partitioning of those words between the exponent and mantissa. Variants for complex arithmetic in *MPL* are also to be added.

For all the real number representations to be implemented, complex arithmetic will be implemented both in its conventional (real and imaginary part) form and in modulus-argument (or polar) form. Appropriate elementary and special function routines will also be available for each of these data types.

Much work has, of course, been done over the years on various aspects of the floating-point system. This has included the IEEE standards, hardware algorithm development, error analysis and correction, CORDIC algorithms for elementary functions and multiple precision packages. (See [29]-[38], for example.)

Other variations on the basic floating-point arithmetic which are included are implementations of directed rounding so that interval arithmetic [39]-[42] may be simulated along with conventional arithmetic operations. In this context a "super-accumulator" for "exact" accumulation of floating-point inner products is to be implemented using the processor array to simulate the multiple precision unit.

The extended floating-point systems of Matsui-Iri [74] and Hamada [71], [72] and [76] are based on the principle of only using the necessary number of bits in a floating-point word to represent the exponent. These are therefore developments of Morris's tapered floating-point system [75]. The intention of both of these systems is to alleviate the overflow/ underflow problem of floating-point arithmetic.

Matsui and Iri used part of the computer word to represent a pointer which indicates the number of bits allocated to the exponent with the rest then being available for mantissa representation. The relative representation error therefore grows with the magnitude of the number being represented - approximately linearly with the logarithm of its binary exponent. However, a "single precision" version of this representation requires 5 bits for this pointer and so can only yield higher precision over a very restricted

range. The system is therefore suitable only for longer wordlengths.

This is also true of Hamada 's "Universal Representation of Real Numbers" or URR in which Matsui and Iri's pointer is replaced by a dual purpose segment of the representation. In essence, this section of the word replaces both the pointer and the first bit of the exponent. Thus if the exponent has the form 2^m+n the first bit is replaced by a unary string of m bits followed by a terminator. The rest of the exponent (the binary representation of n) occupy the next m bits and these are followed by the mantissa. Because of the need for the terminating bit in the representation of m, it follows that this representation is less compact than Matsui & Iri's once m is greater than the pointer length of the latter representation.

The computer arithmetic laboratory will include both 32- and 64-bit versions of both these arithmetics as further variations on the binary floating-point system.

III.2.2 Logarithm-based arithmetics

Logarithmic arithmetic has been extensively studied in recent years as an alternative to floating-point for real arithmetic. Work has included theoretical error analysis studies, algorithmic analysis and developments and practical hardware processor designs. (See [43]-[48] for a sample of this work.)

The basis of logarithmic arithmetic is that a positive number is represented by its base 2 logarithm. This logarithm is represented in fixed-point form. The internal arithmetic of the logarithmic arithmetic in the MP-1 laboratory is therefore one of the places where the fixed-point binary fraction arithmetic referred to in Section III.1.2 is used.

The recently developed algorithms based on polynomial interpolation techniques [47] will be incorporated into the implementation.

It is easy to extend the ideas of logarithmic arithmetic to an arbitrary base. Using e the base of natural logarithms may have some advantages for logarithmic complex arithmetic and for the evaluation of elementary functions within this system. This, too, will be added to the laboratory.

Natural logarithmic arithmetic is a bridge to the implementation of the level-index, LI, and symmetric level-index, SLI systems [54]-[70]. The implementation of these systems is discussed in greater detail in the next section.

IV. SLI implementation

Like many arithmetic systems the LI and SLI systems rely on a simpler arithmetic for their underlying

internal arithmetic. In this case the underlying arithmetic is fixed-point fraction arithmetic. This section begins with a brief description of this and then of the LI and SLI implementations.

IV.1 Fraction arithmetic

In the fraction arithmetic of the MP-1 computer arithmetic laboratory, a number f with $|f| < 1$ is represented by a sign digit followed by a number of fraction digits. Each of these is a hexadecimal digit (or nibble) which simplifies spreading an arithmetic operation across the processor array.

The sign digit can obviously carry much more information than just the sign of the number. This additional space allows the storage of a reciprocation bit (or flag), and an overflow indicator bit. The reciprocation bit allows meaningful results to be returned for division of a larger number by a smaller one. If this result is itself to be used later as a divisor, unnecessary failure is thus averted.

Similarly, the "overflow bit" can be used to prevent overflow resulting from the addition of two fractions. In fact two such bits are available and these could be used to extend the representable range to (-4, 4). Adding further integer nibbles can obviously extend this range.

Fractions of up to 15 nibbles, can be stored in the standard MPL data type `long long` - a 64-bit integer which is one of its extensions of ANSI C. There are therefore packing and unpacking routines for conversion between types such as `fraction10` (a fraction with sign plus 10 hexadecimal digits) and its various components. The bit manipulation operators of C make this operation reasonably straightforward. Further conversion routines are provided for changing between conventional real storage and the fraction types.

The available types will allow up to 15 hexadecimal digit fractions. Longer fractions can be simulated by using more than one word - or, more likely, by using more than one processor - for its storage.

Once the storage of such quantities is achieved, addition and subtraction are implemented by using their integer counterparts. The same is not true of multiplication.

Overflow (or wraparound) of integer multiplication is not appropriate since the most significant digits of the product are the ones which correspond to these conditions. However the hexadecimal digit products can be constructed using unsigned 8-bit integer arithmetic and then combined with appropriate shifts to reformulate the result. Similarly the hex digits provide a natural framework for a (software) radix-16 nonrestoring division algorithm.

The presence of the reciprocation bit necessitates a preprocessing of fractions for multiplication and/or division so that the correct sign and reciprocation sign are assigned to the result of the appropriate final

arithmetic operation. For example division of a larger fraction, x, by a smaller one, y, is performed by setting the reciprocation bit of the result and computing the reciprocal quotient y/x.

Many of the decision processes here are reminiscent of those used in the Turbo Pascal implementation of SLI arithmetic described in [68]-[70].

IV.2 LI arithmetic

In the LI system a positive number X is represented by its generalized logarithm x where

$$X = \phi(x) \tag{1}$$

and the generalized exponential function ϕ (the inverse of the generalized logarithm) is given by

$$\phi(x) = \begin{cases} x & 0 \le x \le 1 \\ \exp(\phi(x-1)) & x \ge 1 \end{cases} \tag{2}$$

The basic representation. arithmetic algorithms and analysis for this system were discussed in detail in [54]-[57], [61], [64].

To give a flavor of the MP-1 implementation of this system we describe just the addition algorithm and its use of the fixed-point fraction arithmetic. This operation consists of finding z such that

$$\phi(z) = \phi(x) \pm \phi(y) \tag{3}$$

where $x = l + f > m + g = y > 0$ and $l = [x]$, $m = [y]$. This is achieved by computing members of the sequences

$$a_j = \frac{1}{\phi(x-j)}, \quad b_j = \frac{\phi(y-j)}{\phi(x-j)}, \quad c_j = \frac{\phi(z-j)}{\phi(x-j)} \tag{4}$$

The first two of these are evaluated by similar recurrence equations for *decreasing* values of j:

$$\begin{aligned} a_{-1} &= \exp\left(\frac{-1}{a_j}\right) & a_{l-1} &= e^{-f} \\ b_{j-1} &= \exp\left(\frac{-1+b_j}{a_j}\right) & b_{m-1} &= a_{m-1}e^{-g} \end{aligned} \tag{5}$$

The initial value for the b-sequence can be redefined to allow the simultaneous computation of these two sequences. Their values are bounded by 0 and 1 and the analysis of the algorithm [56] shows that they can be computed to fixed absolute precisions. It follows that fixed-point fractions are the desired internal arithmetic form.

The remainder of the algorithm consists of setting

$$c_0 = 1 \pm b_0 \qquad (6)$$

then computing terms of the c-sequence by another short recurrence, and performing a final step to obtain z. The c_j's are bounded, by [0, 1] for subtraction and [1, 2] for addition. Again, fixed-point fraction arithmetic is appropriate.

The analysis of the LI arithmetic algorithms [56] shows that, for a 32-bit LI wordlength, the data types `fraction10` and `fraction8` (that is fractions with 10 and 8 hexadecimal digits) are suitable for the computation of the a-sequence and the b- and c-sequences respectively. Furthermore, the sign nibble of the fraction representation above admits a 1-bit integer part so that the terms of the c-sequence for addition create no difficulty.

Efficient computation with these data types will certainly require implementation of special algorithms for the exponential and logarithm functions for the restricted range of arguments which are encountered in the LI algorithms. These special algorithms can be spread across the processor array. They would probably be based on the modified CORDIC algorithms originally presented in [66] or the table-lookup approach of [65].

Development of these algorithms is another task which will be eased by the computer arithmetic laboratory.

IV.3 SLI arithmetic

We begin with a brief description of a new SLI arithmetic algorithm and then consider its implementation in the MP-1 computer arithmetic laboratory. The notation here is the same as for LI arithmetic above except that now a real number X is represented by

$$X = \pm \phi(x)^{\pm 1}$$

with ϕ given by (2) and $x \geq 1$.

IV.3.1 Modified SLI algorithm

In the standard SLI arithmetic algorithms described in [56] and [58] all the basic arithmetic operations involve the computation of a quantity c_0 from which computation of the c-sequence proceeds.

For the "large" case, the add/subtract operation is just the LI operation (3) above. Then c_0 is given by

$$c_0 = 1 \pm b_0 = 1 \pm \frac{\phi(y)}{\phi(x)} \qquad (7)$$

The corresponding "mixed" operation is

$$\Phi(z) = \Phi(x) \pm \Phi(y)^{-1}$$

with c_0 given by

$$c_0 = 1 \pm a_0 \alpha_0 = 1 \pm \frac{1}{\Phi(x)\,\Phi(y)} \tag{8}$$

For "small" arithmetic the basic operation is

$$\Phi(z)^{-1} = \Phi(x)^{-1} \pm \Phi(y)^{-1}$$

with $c_0' = 1/c_0$ given by

$$c_0' = 1 \pm \beta_0 = 1 \pm \frac{\Phi(x)}{\Phi(y)} \tag{9}$$

There are similar recurrence relations to those in (5) which are used from appropriate starting values to generate the members of the α- and β-sequences given by

$$\alpha_{j-1} = \exp\left(\frac{-1}{\alpha_j}\right) \qquad (j = m-1, \ldots, 1)$$

$$\beta_{j-1} = \exp\left(\frac{-1+\beta_j}{a_j \beta_j}\right) \qquad (j = l-1, \ldots, 1)$$

where, again, l, m are the levels of x, y respectively. Note that in all cases, the first argument to the arithmetic operation is assumed to be the larger in absolute value so that $x \geq y$ for the large case and $x \leq y$ in the small case.

These arithmetic operations are analysed in [58] in terms of the required precisions in the fixed-point computation of the sequences in order to deliver results with error comparable with inherent errors.

The alternative algorithms presented here are based on using only the a- and α-sequences. This has great potential advantages for both SIMD software and VLSI hardware implementation of SLI arithmetic since the definitions of these sequences are *identical* for the two arguments x and y.

These alternative algorithms reduce to redefining the initial values of the c-sequences by:

$$c_0 = 1 \pm \frac{a_0}{\alpha_0} \qquad \text{large arithmetic} \tag{10}$$

$$c_0 = 1 \pm a_0\,\alpha_0 \qquad \text{mixed arithmetic} \tag{11}$$

and

$$c_0' = 1 \pm \frac{\alpha_0}{a_0} \qquad \text{small arithmetic} \tag{12}$$

in place of (7)-(9). The remainder of the algorithm remains unchanged. We observe here that the divisions in (10) and (12) are always of a smaller quantity by a larger so that our fixed-point fraction arithmetic remains appropriate.

The precision requirements of the fixed-point internal computation will, of course, be slightly different for this modified algorithm. The detailed error analysis of this algorithm will be published elsewhere. The availability of variable wordlength fixed-point fractions will simplify computational testing of this algorithm.

Extensions of this algorithm to the extended arithmetic operations such as summation, scalar products and vector norm computations (see [62], [69] for example) yield further simplifications in the algorithm logic and therefore in the potential for VLSI hardware designs. A SIMD software implementation is a natural step in this direction.

IV.3.2 SLI Implementation

In this subsection we highlight some of the features of the MP-1 implementation of SLI arithmetic with reference to the task of summing a series of SLI terms which fits the processor array.

This example demonstrates some of the simplifications which follow from the adoption of the revised SLI algorithm described above. It is also a good vehicle for illustrating some of the features of the MPL language and its extensions of ANSI C. One of the primary benefits of this from the arithmetic viewpoint is that the SIMD instructions make it plain where there is multiple use of the same instruction which is a good indicator of suitability for VLSI design. The many reduction algorithms that are built into the language also show clearly the places in a VLSI algorithm where adder, or other logic, trees would be used.

These advantages obviously carry over to any arithmetic system that is to be implemented on this or any similar SIMD architectures.

First the single precision, 32-bit, SLI data type `slisingle` can be identified with the 32-bit integer type `long` in such a way that the integer ordering is the correct SLI ordering. This is just the same data packing routine as was used in [68]-[70]. This order-preserving mapping is important for the identification of the largest element of the array of terms.

These terms would exist as a variable X of type `plural slisingle` which is to say it has one

instance on each of the processors in the 64×64 array.

To describe the algorithm we shall denote the individual terms by $X_i = s_i \phi(x_i)^{r_i}$ ($i = 0, 1, \ldots, 4095$). The largest element in this array of terms, and, more importantly, its position can be obtained using the built-in MPL reduction functions `reduceMax32` and `rank32`. We shall denote the position of the maximal element by p. For simplicity we shall assume $|X_p| \geq 1$ so that $r_p = 1$.

The next step of the algorithm is to compute the a-sequence for each term. This operation is performed simultaneously on each processor to produce a `plural fraction10 a[7]` where again the word "plural" indicates the existence of this array on all processors. (The dimension 7 here reflects the maximum level needed in SLI arithmetic.) We shall denote the values of a[0] by A_i.

The only branch in the algorithm is now used to compute the quantities

$$B_i = \begin{cases} s_p s_i A_p / A_i & \text{if } r_i = +1 \\ s_p s_i A_p A_i & \text{if } r_i = -1 \end{cases} \tag{13}$$

These terms are then summed over all processors to obtain c_0 using the fraction equivalent of the built-in `reduceAdd` function. The number of terms demands that a maximum of 12 bits, or 3 hexadecimal digits, are needed for the integer part of c_0.

The computation is completed by generating subsequent members of the c-sequence as for regular SLI addition.

The algorithm just described is not only much simpler than that presented in [69]. The use of the parallel instructions and reduction-based algorithms demonstrates clearly the inherent suitability of the algorithm for VLSI implementation.

The underlying fraction arithmetic requires just a few extensions beyond regular arithmetic operations. For example, a special purpose routine for computing $\exp(-1/F)$ for a fixed-point fraction F in (0, 1) to a fixed absolute precision is needed to compute the various a-sequences efficiently. This can be achieved using a modified CORDIC algorithm similar to those in [66] and [69].

V. Conclusions

In this paper we have introduced the ideas behind the development of a software computer arithmetic laboratory on a massively parallel SIMD array processor. The particular machine used is a DEC/ MasPar MP-1 with 4096 processors although the principles would apply equally well on any other similar SIMD

machine.

A wide variety of number representation and arithmetic systems for computers can be incorporated into this laboratory. This paper has described some of those and then presented some salient details of just a few, including fixed-point fractions and the level-index and symmetric level-index systems.

The primary benefits to be gained are in the provision of a reasonable basis for comparison between various arithmetic forms and in allowing algorithmic experimentation as an aid to hardware design processes.

VI. Bibliography

GENERAL

[1] K.Hwang, *Computer Arithmetic: Principles, Architecture and Design* John Wiley & Sons, 1978

[2] K.Hwang & F.A.Briggs, *Computer Architecture and Parallel Processing* McGraw-Hill, 1984

[3] N.R.Scott, *Computer Number Systems and Arithmetic* Prentice Hall, 1985

[4] P.Sterbenz, *Floating-point computation* Prentice Hall, 1974

[5] E.E.Swartzlander, *Computer Arithmetic* Dowden, Hutchinson & Ross, Stroudsburg, PA, 1980

[6] E.E.Swartzlander, *Computer Arithmetic, Vol. II* IEEE Computer Society Press,1990

[7] S.Waser & M.J.Flynn, *Introduction to Arithmetic for Digital Systems Designers* Holt, Reinhart & Winston, 1982

FIXED-POINT ARITHMETIC

[8] A.Aviziensis, *Signed-digit number representations for fast parallel arithmetic* IRE Trans Elec Comp, 10 (1961) 389-400

[9] J.Duprat, Y.Herreros & S.Kla, *New redundant representations of complex numbers and vectors* ARITH10, pp. 2-9, IEEE Computer Society, Washington DC 1991

[10] M.D.Ercegovac, T.Lang & P.Montuschi, *Very high radix division with selection by rounding and prescaling* ARITH11, pp 112-129, IEEE Computer Society, 1993

[11] J-M.Muller, *Arithmetique des Ordinateurs* Masson, Paris, 1989

[12] G.S.Taylor, *Radix 16 SRT dividers with overlapped quotient selection stages* ARITH7 pp. 64-71, IEEE Computer Society, Washington DC, 1985

[13] D.C.Wong & M.J.Flynn, *Fast division using accurate quotient approximations to reduce the number of iterations* IEEE Trans Comp 41 (1992) 981-995

RESIDUE NUMBER SYSTEMS

[14] S.S.Bizzan, G.A.Jullien, N.M.Wigley & W.C.Miller, *Integer mapping architectures for the polynomial ring engine* ARITH11, pp 44-51, IEEE Computer Society, Washington DC, 1993

[15] W.A.Chren, *A new residue number system division algorithm* Comput. Math. Appl 19 (1990) 13-29

[16] G.I.Davida & B.Litow, *Fast parallel arithmetic via modular representation* SIAM J Computing 20 (1991) 756-765

[17] D.Gamberger, *New approach to integer division in residue number systems* ARITH10, pp. 84-9, IEEE Computer Society, Washington DC 1991.

[18] R.T.Gregory & D.W.Matula *Base conversion in Residue Number Systems* ARITH3, pp117-125, IEEE Computer Society, Washington DC 1975

[19] M.Griffin, M.Sousa & F.J.Taylor *Efficient scaling in the residue number system* Proc IEEE Conf. on ASSP, IEEE, New York, 1989

[20] M.A.Hitz & E.Kaltofen, *Integer division in residue number systems* Comp Sci Tech Rep #93-9, Rensselaer Polytechnic Institute, May 1994

[21] B.J.Kirsch & P.R.Turner, *Adaptive beamforming using RNS arithmetic* ARITH11, pp. 36-43, IEEE Computer Society, Washington DC, 1993

[22] Mi Lu & J-S.Chiang, *A novel division algorithm for the residue number system* IEEE Trans Comp 41 (1992) 1026-1032.

[23] J.Mellot, J.Smith, & F.J.Taylor, *The Gauss Machine: A Galois-Enhanced Quadratic Residue Number System systolic array* ARITH11, pp 156-162, IEEE Computer Society, Washington DC, 1993

[24] K.H.O'Keefe *A note on fast base-extension for Residue Number Systems with three moduli* IEEE Trans Comp 22 (1975) 1132-1133

[25] M.A.Soderstrand, W.K.Jenkins, G.A.Jullien & F.J. Taylor *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, IEEE, New York, 1986

[26] N.Szabo & R.Tanaka *Residue Arithmetic and its Application to Computer Technology*, McGraw-Hill, 1967

[27] P.R.Turner *An improved RNS division algorithm* NAWC-AD, Warminster, Tech Report 1994 (submitted to ARITH12)

[28] N.M.Wigley, G.A.Jullien & D.Reaume *Large dynamic range computations over small finite rings* IEEE Trans Comp 43 (1994) 78-86

FLOATING-POINT, IEEE

[29] J-C.Bajard, S.Kla & J-M.Muller, *BKM: A new algorithm for complex elementary functions*

ARITH11, pp. 146-153, IEEE Computer Society, Washington DC 1993

[30] R.Brent, *A Fortran multiple precision arithmetic package* ACM Trans Math Software 4 (1978) 57-70

[31] M.Daumas & D.W.Matula, *Design of a fast validated dot product operation* ARITH11, pp. 62-69, IEEE Washington DC, 1993

[32] J.W.Demmel & X.Li, *Faster numerical algorithms through exception handling* ARITH12, pp. 234-241, IEEE Computer Society, Washington DC 1993

[33] M.D.Ercegovac & T.Lang *On-the-fly rounding* IEEE Trans Comp 41 (1992) 1497-1503

[34] G.J.Hekstra & E.F.A.Deprettere, *Floating-point CORDIC* ARITH11, pp. 130-137, IEEE, Computer Society, Washington DC, 1993

[35] *IEEE Standard for binary floating-point arithmetic*, ANSI/ IEEE Std 754, IEEE, New York,1985

[36] D.M.Priest, *Algorithms for arbitrary precision floating-point arithmetic* ARITH11, pp.132-143, IEEE Computer Society, Washington DC, 1991

[37] M.Schulte & E.Swartzlander, *Exact rounding of certain elementary functions* ARITH11, pp. 138-145, IEEE Computer Society, Washington DC 1993

[38] E.M.Schwartz & M.J.Flynn, *Hardware starting approximation for the square-root operation* ARITH11, pp. 103-111, IEEE Computer Society, Washington DC, 1993

INTERVAL ARITHMETIC & SUPER ACCUMULATORS

[39] O.Aberth & M.J.Schaefer, *Precise computation using range arithmetic via C++* ACM Trans Math Soft 18 (1992) 481-491

[40] A.Knofel, *Fast hardware units for the computation of accurate dot products* ARITH10, pp. 70-74, IEEE Computer Society, 1991

[41] U.W.Kulisch &W.L.Miranker, *Computer Arithmetic in Theory and Practice* Academic Press, 1981

[42] M.Muller, C.Rub & W.Rulling, *Exact accumulation of floating-point numbers* ARITH10, pp. 64-69, IEEE Computer Society, Washington DC, 1991

LOGARITHMIC ARITHMETIC

[43] M.G.Arnold, T.A.Bailey, J.R.Cowles & J.J.Cupal, *Redundant logarithmic arithmetic* IEEE Trans Comp 39 (1990) 1077-1086

[44] M.G.Arnold, T.A.Bailey, J.R.Cowles & M.D.Winkel, *Applying features of IEEE 754 to signed logarithm arithmetic* IEEE Trans Comp 41 (1992) 1040-1050.

[45] J.L.Barlow & E.H.Bareiss, *On roundoff distribution in floating-point and logarithmic arithmetic* Computing 34 (1985) 325-364

[46] D.M.Lewis, *An architecture for addition and subtraction of long word length numbers in the logarithmic number system* IEEE Trans Comp 39 (1990) 1326-1336

[47] D.M.Lewis, *An accurate LNS arithmetic unit using interleaved memory function interpolator* ARITH11, pp. 2-9, IEEE Computer Society, Washington DC 1993

[48] D.M.Lewis & L.K.Yu, *Algorithm design for a 30 bit integrated logarithmic processor* ARITH9, pp. 192-199, IEEE Computer Society, Washington DC, 1989

LEXICOGRAPHIC CONTINUED FRACTIONS

[49] P.Kornerup & D.W.Matula, *Finite precision rational arithmetic: An arithmetic unit* IEEE Trans Comp 32 (1983) 378-387

[50] P.Kornerup & D.W.Matula, *Finite precision lexicographic continued fraction number systems* ARITH7, pp. 207-214, IEEE Computer Society, Washington DC, 1985

[51] P.Kornerup & D.W.Matula, *An on-line arithmetic for bit-pipelined rational arithmetic* J Parallel & Dist Comp 5 (1988) 310-330

[52] P.Kornerup & D.W.Matula, *Exploiting redundancy in bit-pipelined rational arithmetic* ARITH9, pp. 119-126, IEEE Computer Society, Washington DC, 1989.

[53] D.W.Matula & P.Kornerup, *An order-preserving finite binary encoding of the rationals* ARITH6, pp. 201-209, IEEE Computer Society, Washington DC, 1983

LI/ SLI ARITHMETIC

[54] C.W.Clenshaw, D.W.Lozier, F.W.J.Olver & P.R. Turner, *Generalized exponential and logarithmic functions* Comp Math Applics 12B (1986) 1091-1101

[55] C.W.Clenshaw & F.W.J.Olver, *Beyond floating point* J ACM 31 (1984) 319-328

[56] C.W.Clenshaw & F.W.J.Olver, *Level-index arithmetic operations* SIAM J Num Anal 24 (1987) 470-485

[57] C.W.Clenshaw, F.W.J.Olver & P.R.Turner, *Level-index arithmetic: An introductory survey* Numerical Analysis and Parallel Processing, Springer, 1989, pp 95-168

[58] C.W.Clenshaw & P.R.Turner, *The symmetric level-index system* IMA J Num Anal 8 (1988) 517-526

[59] C.W.Clenshaw & P.R.Turner, *Root-squaring using level-index arithmetic* Computing 43 (1989) 171-185

[60] D.W.Lozier, *An underflow-induced graphics failure solved by SLI arithmetic* ARITH11, pp 10-17, IEEE Computer Society, Washington DC, 1993

[61] D.W.Lozier & F.W.J.Olver, *Closure and precision in level-index arithmetic* SIAM J Num Anal 27

(1990) 1295-1304

[62] D.W.Lozier & P.R.Turner, *Robust parallel computation in floating-point and sli arithmetic* Computing 48 (1992) 239-257

[63] D.W.Lozier & P.R.Turner, *Symmetric level-index arithmetic in simulation and modeling* J Res NIST 97 (1992) 471-485

[64] F.W.J.Olver, *Rounding errors in algebraic processes - in level-index arithmetic* In Reliable Numerical Computation (M.G.Cox and S.Hammarling eds.) pp 197-205, Oxford University Press, 1990

[65] F.W.J.Olver & P.R.Turner, *Implementation of level-index arithmetic using partial table lookup* ARITH8, pp 144-147, IEEE Computer Society, Washington DC 1987

[66] P.R.Turner, *Towards a fast implementation of level-index arithmetic* Bull IMA 22 (1986) 188-191

[67] P.R.Turner, *Algorithms for the elementary functions in level-index arithmetic* Scientific Software Systems (M.G.Cox & J.C.Mason, eds) Chapman & Hall, 1990, pp 123-134

[68] P.R.Turner, *A software implementation of sli arithmetic* ARITH9, pp 18-24, IEEE Computer Society, Washington DC 1989

[69] P.R.Turner, *Implementation and analysis of extended SLI operations* ARITH10 pp. 118-126, IEEE Computer Society, Washington DC 1991

[70] P.R.Turner, *Complex SLI arithmetic: Representation, algorithms and arithmetic* ARITH11 pp. 18-25, IEEE Computer Society, Washington DC, 1993

EXTENSIONS OF FLOATING-POINT

[71] H.Hamada, *URR: Universal representation of real numbers* New Generation Computing 1 (1983) 205-209

[72] H.Hamada, *A new real number representation and its operation* ARITH8, pp. 153-157, IEEE Computer Society, Washington DC 1987

[73] T.E.Hull, M.S.Cohen & C.B.Hall, *Specifications for a variable-precision arithmetic coprocessor* ARITH10, pp. 127-131, IEEE Computer Society, 1991

[74] S.Matsui & M.Iri, *An overflow/ underflow free floating-point representation of numbers* J Inform Process 4 (1981) 123-133 (Also reproduced in [4b])

[75] R.Morris, *Tapered floating-point: A new floating-point representation* IEEE Trans Comp 20 (1971) 1578-1579

[76] H.Yokoo, *Overflow/Underflow-free floating-point number representations with self-delimiting variable length exponent field* ARITH10, pp. 110-117, IEEE Computer Society, Washington DC 1991